Go语言博客实践

wizardforcel

Published
with GitBook

目錄

介绍	0
第1章：选择 Martini 框架	1
第2章：永远的MVC	2
第3章: 面向对象和并发	3
第4章：服务器裸奔	4
第5章: 静态博客	5
第6章: 解析器与舞台剧	6
第7章: 解析器与ABNF	7
第8章: Rivet	8

Go语言博客实践

> 作者：achun
>
> 来源：Go-Blog-In-Action

(Martini 版本) 老版本点击这里

Go Blog In Action 中文名 **Go**语言博客实践. 是对 TypePress 开发过程中的想法, 方法, 探讨等任何方面同步整理成的电子书.

作者: 喻恒春

目录

- 第1章：选择 Martini 框架
  - 第1节：为什么是Martini
  - 第2节：Package选择与修改
  - 第3节：依赖注入
- 第2章：永远的MVC
  - 第1节：永远的MVC
  - 第2节：常见的方法
  - 第3节：TypePress的方法
- 第3章: 面向对象和并发
  - 第1节：面向对象
  - 第2节：并发下维护上下文
  - 第3节：Martini下的并发
- 第4章：服务器裸奔
  - 第1节：配置基本参数
  - 第2节：基本功能
  - 第3节：模块化
- 第5章: 静态博客 未完待续, 穿插解析器部分
- 第6章: 解析器与舞台剧
  - 第1节：汤姆的故事
  - 第2节：PEG
- 第7章: 解析器与ABNF
  - 第1节: ABNF

    - 第2节: 四则运算表达式
    - 第3节: 解析器
    - 第4节: 手工至上
- 第8章: Rivet
    - 第1节: Router
    - 第2节: Rivet
    - 第3节: Module

授权许可

除特别声明外，本书使用CC BY-SA 3.0 License（创作共用 署名-相同方式共享3.0许可协议）授权。

为什么是Martini

在上一版 Go语言博客实践 中, 作者提到不使用框架来完成一个 Blog 系统. 现在选择 Martini 作为基础框架确实和 Martini 设计的独特性有关. Martini 的核心 Injector 实现了依赖注入 (参见 控制反转).

这里有两篇博客可供参考 Martini的工作方式 和 Martini中的Handler. 简单的说 Injector 通过 reflect 削弱了合作对象间引用依赖.

对于 Martini 的使用可以简单总结为:

- Martini 对象方法 Map/MapTo/Use/Handlers/Action 非并发安全, 服务器运行前使用.
- Router 对象也是非并发安全的, 服务器运行前使用.
- Context 对象是在 http Request 时动态创建的.
- 所有要使用的对象必须先 Map/MapTo.
- 对 http.ResponseWriter 任何的 Write 都会完结响应. 内部方法是终止了响应 Handler.
- 善用 Context 对象的 Next 方法会产生奇效.

上一版本因为不能找到 "解耦" 的框架而放弃使用框架. Martini 在 Injector 的支持下为"解耦"提供了可能. 这正是笔者希望的.

Package选择与修改

- Martini社区 martini-contrib

  Martini 社区贡献的 package, 可能会使用一些.如果您研究了 Martini 和这些 contrib package, 您会发现真的解耦了.

- 角色控制 accessflags

  角色控制是应用中的常见需求, accessflags 基于 Martini 实现了一个通过 interger 标记值控制 Martini.Handler 是否允许访问. 可以用于角色控制. (已被社区收录)

- 配置文件支持 tom-toml

  笔者重新写了一个 TOML 解析器 tom-toml, 参见文章有关tom-toml的一些事儿, 和第六章的内容.

- 数据库操作 typepress/db

  upper.io/db 是 gosexy/db 的重构版本. 代码质量很高. 但是包路径问题同样给 import 造成了问题. 为方便, 笔者 fork 了一个 github 版本 typepress/db. upper.io/db 为常见的 SQL/NoSQL 数据库提供了统一的调用接口, 这是非常难能可贵的.

- 日志支持 typepress/log

  typepress/log 学习了 uniqush/log 的一些好想法重新构建的. typepress/log 支持日志分割, 并实现了一个 file 日志, 一个 email 日志.

- template 模板

  可能会有几个备选版本 martini-contrib 中有render, 笔者写有 template.

- 国际化支持 i18n

  这是一个简洁的 i18n 支持接口, 仿照 fmt.Sprint, fmt.Sprintf 的形式. 在使用中即便暂时没有国际化支持的需求, 使用 i18n 所带来的消耗也是极小的. 完全可以当作 fmt.Sprint, fmt.Sprintf 使用.

依赖注入

Martini 的核心就是实现依赖注入, 高度解耦. 依据依赖注入的思路, 上述的 package 被替换掉应该不是一件复杂的事情. 随时引入依赖注入也应该很容易. 也许吧, 实践中我会关注这个事情.

永远的MVC

MVC 是对软件软件系统三个基础部分的描述, 就好像冯·诺伊曼结构 或者哈佛结构对计算机体系结构的定义. MVC 是客观存在的, 是事实.

- 无论你的代码中是否显示的使用了 MVC 的方法, 她都存在.
- 无论你的代码是否显示的遵循 MVC 的方法, 她都存在.
- 无论你的代码是否违背了公认的 MVC 方法, 她都存在.

总之只要你写代码, 无论你怎么写, 她都存在. 对于一个运算表达式:

```
a := b + c
```

又或者对于一个函数:

```
func Foo(b,c interface{})(a interface{}){
    // 函数主体, 完成的就是控制了
}
```

- a 就是 view
- "+" 就是 controller
- b,c 就是model

MVC在心中

常见的方法

开发者通常会这么做:

- 类型声明, 包含 `Controller` 这个词
- 文件名, 包含 `Controller` 这个词
- 目录名, 包含 `Controller` 这个词

在名称上显示出来是好方法. 这增强了代码可读性, 一目了然. 当然如果目录名已经用了 `Controller` 了, 目录之下的文件或者类型声明是否有必要再加上 `Controller` , 语言不同, 习惯不同, 并没有定式. Go 语言一向提倡 能省则省.

依据 MVC 目录看起来是这个样子

```
├──conf
├──controllers // 这里面因归属关系又嵌套了 N 层
│   └──something
├──models
└──views
    ├──login
    └──signup
```

典型的树状结构增强了代码可读性, 是常见 MVC 目录组织形式.

TypePress的方法

能保有树状代码目录结构无疑有助于管理维护. 基于 Martini Injector 风格下, 对象间的依赖被降低, 对象依赖关系不必遵循树状结构, 目录结构也不必保持树状. 这在很多时候会更灵活, 同时这也是一种不常见的方法, TypePress 将尝试使用一些.

扁平目录

意思是尽量降低目录曾经深度, 视觉上不显示依赖关系. 事实上这类似于组件独立化. 如果代码是辅助性的, 例如服务器端的 Handler, 那就表现为独立的 Rep. 如果可以分组, 例如浏览器前端组件, 那就在同一个 Rep 下做扁平目录.

自动注册路由

实验性想法, 目的是给应用生成工具提供基础支持. 对于具体应用, 比如博客, 业务层面的控制器, 多具有层级关系. 其中还涉及角色控制和 http Request Method. 用 martini.Router 写起来像这样

```
router.Get("/profile", roleAllow("Admin"),youHandler)
```

如果用自动注册路由写起来像这样

```
core.AutoRouter(youHandler)
```

前提是要把 path,method,role 写到文件路径里. 对应上面的例子, 完整的运行期路径写起来有这样几种

- github.com/UserName/RepName/Admin/GET/profile.go
- github.com/UserName/RepName/Admin/GET.profile.go
- github.com/UserName/RepName/Admin.GET.profile.go

都是能被识别的写法, 依据大小写和"/","."作为分割符号实现自动注册路由是可能的. 由此设计出自动构建/装配工具就有了基础.**TypePress** 将尝试这种方式.

面向对象

OOP 的思想, 无疑是非常实用有效的. 事实是, 无论语言是否直接支持面向对象的编程. 程序员在写代码的时候常常会应用 OOP 的思想.

Go 语言下没有类(Class), 没有构造函数, 没有 `this` 指针, 没有多态, 只有复合对象(或匿名属性). 复合对象和继承是完全不同的. 在以后的文字中, 继承这个词不再代表一般 OOP 下的继承, 指的是复合对象. 应用 OOP 的思想, WEB 应用下控制器常见形式祖先类型的示意写法(现实中没有太大意义).

```go
// 定义基础控制器结构
type BaseController struct {
	Data interface{}          // 应用要维护的数据
	Req  *http.Request        // 请求对象
	Res  http.ResponseWriter // 响应对象
}

// 官方 net/http 包要求实现的接口
func (p *BaseController) ServeHTTP(w http.ResponseWriter, r *http.Request) {
	p.Req = r // 保存起来供实例使用
	p.Res = w
	if r.Method == "POST" {
		p.Post()
	}
}

// 对应 http POST 方式
func (p *BaseController) Post() {
	// 继承者必须覆盖这个方法, 否则认为不允许 POST 访问
	// BaseController 是不可能知道继承者要做什么, 那就只能返回 403 拒绝访问
	p.Res.WriteHeader(403)
}

// Login 控制器
type Login struct {
	BaseController // 匿名复合
}

// 这里必须覆盖 BaseController.Post, 以实现 Login 的具体行为
func (p *Login) Post() {
	if p.Req.Form.Get("login_name") == "" {
		p.Data = "无效的登录名"
		return
	}
	// 这里省略登录成功的过程
	p.Data = "登录成功"
}

// 把这些行为定义成接口
type Controller interface {
	ServeHTTP(http.ResponseWriter, *http.Request)
	Post()
}
```

用例

```go
http.Handle("/login", &Login{})
```

但是这有并发问题

并发下维护上下文

很明显现实中这样的用法是错误的，因为 WEB 的请求是并发的，这样写所有并发的请求都由同一个 `&Login{}` 去处理响应，`Req`，`Res`，`Data` 在并发中都被指向相同的对象。这是无法正常工作的。这就是常说的维护上下文，Context.

并发环境每一个请求都要有维护独占数据的能力。除非没有独占数据要维护。

先重新审视 官方包 server.go 中的代码

```
type Handler interface {
    ServeHTTP(ResponseWriter, *Request)
}

func Handle(pattern string, handler Handler) { DefaultServeMux.Handle(pattern, handler) }

func HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
    DefaultServeMux.HandleFunc(pattern, handler)
}

func (mux *ServeMux) HandleFunc(pattern string, handler func(ResponseWriter, *Request)) {
    mux.Handle(pattern, HandlerFunc(handler)) // 进行了转换，只是为了符合接口要求
}

type HandlerFunc func(ResponseWriter, *Request) // 确实没有独占数据要维护

// ServeHTTP calls f(w, r).
func (f HandlerFunc) ServeHTTP(w ResponseWriter, r *Request) {
    f(w, r)
}
```

这个 `http.Handler` 接口其实只是被当作一个函数使用了。并发问题留给使用者自己解决。可以这样做

```
http.HandleFunc("/login", func(w http.ResponseWriter, r *http.Request) {
    p := &Login{}
    p.ServeHTTP(w, r)
})
```

每次请求都有新的 `Login` 对象产生。当然这个写法很生硬，如果有 100 个控制器，难道还要写 100 个不同的写法?! 可以采用下面的方法。

函数法

不用结构体直接使用函数，所有上下文维护都在函数内部定义成局部变量，局部变量在函数内部是独占的。

```go
func main() {
    http.HandleFunc("/login", login)
}
func login(w http.ResponseWriter, r *http.Request) {
    // 维护的数据是局部变量
    var data interface{}
    var post = func() { // 用闭包函数访问局部变量
        if r.Form.Get("login_name") == "" {
            data = "无效的登录名"
            return
        }
        data = "登录成功"
    }
    post()
}
```

完全就是个函数, 但是并发下, 这完全没有问题. 问题在于如何和其他的模块进行数据沟通, 方法确实有, 比如可以用 gorilla/context. 但是无法想象整个项目都用这种写法.

约定构造函数

Go 没有构造函数的概念的. 没关系我们约定一个. 其他语言常用 `Constructor` , 这里选用 `New` 更符合 Go 风格.

```go
// 给控制器接口增加一个构造函数
type Controller interface {
    New() Controller
    ServeHTTP(w http.ResponseWriter, r *http.Request)
    Post()
}

// 扩充 Login , 实现 New 方法
func (p *Login) New() Controller {
    return &Login{}
}

// 定义一个 http.Handler 接口, 支持构造函数
type HandlerNew struct {
    Controller Controller
}

// http.Handler 接口实现
func (p *HandlerNew) ServeHTTP(w http.ResponseWriter, r *http.Request) {
    c := p.Constructor.New()
    c.ServeHTTP(w, r)
}
```

用例

```go
http.Handle("/login", HandlerNew{new(Login)})
```

用反射Value.New

反射包 `reflect` 中的 `reflect.Value` 有 `New` 方法, 可以动态的构造出一个新对象. 有些框架就是采用了这种方法, 但是用 `Value.New` 只是得到一个空属性对象, 要对对象进行初始化依然要约定初始化函数, 这反而比 约定构造函数费事儿. 这里就不具体讨论了.

Martini下的并发

笔者在发现 Martini 之前也很困惑到底用什么办法更好, 所以写了早期的 TypePress, 和对应的 Go-Blog-In-Action. Martini 巧妙解决了 WEB 并发中的上下文维护. Martini 发现了 WEB 开发中单个请求响应要维护的上下文有这样的事实:

- 数据类型是预知的 很显然
- 数据类型有限的 很显然
- 数据类型常常是唯一的 就算偶有不唯一, 定义个别名就行了, 这很容易
- 阶段响应, 完整的响应过程往往分多个阶段, 为了代码复用, 各个阶段有独立的代码

因此 Martini 采用了这样的方案:

1. Martini 负责动态构建一个 Context 对象, Context 继承自 Injector
2. Martini 的 Handler 是一组 []Hanlder, 有序执行
3. 使用者对 Handler 进行阶段性功能划分, 先执行的负责准备好上下文数据 dat
4. 通过 Map(dat) 保存到 Context. (实际由 Injector 负责)
5. 后续 Hander 要用 dat, 直接在 Handler 函数中加入参数 dat datType
6. Injector 通过 reflect 分析 Handler 的参数类型, 并取出 dat, 调用 Handler

这个方法比 gorilla/context 更高效实用, 虽然都是用 map 保存上下文数据, 差别有

- gorilla/context 的 map 是全局的, Martini 保存到 Context
- gorilla/context 只是做了 key/value 存储, Martini 完成了 Handler 调用

这和 OOP 有何关系? 关系是

```
golang 不是真正的继承, 这给维护上下文数据造成了问题.
Martini 解决了上下文数据维护问题, 应用可以放心的用复合写逻辑代码.
上下文数据交给 Martini 就好.
```

服务器裸奔

在 go 语言下要跑起一个HTTP服务器是很容易的.

```
package main

import "net/http"

func main() {
    http.ListenAndServe(":8080", http.FileServer(http.Dir("/usr/share/doc")))
}
```

这就行了,一个静态文件服务器就跑起来了. TypePress 下的代码是这样的

```
package main

import "github.com/typepress/server"

func main() {
    server.Simple()
}
```

这个服务器没有设置任何 Route, 只返回 404, 裸奔的服务器. 这只是表面, 下面来列举下这个 Simple Server 背后都做了哪些工作

配置基本参数

有三种方法设置服务器基本参数:

- 通过命令行参数 --help 可以获得帮助列表
- os.Getenv 获取 应用通过 os.Setenv 设置参数
- 从 TOML 文件读取 TOML 文件支持已经默认加入

基本功能

有些基本的功能是一个框架需要提供的

- 安全关闭机制 shutdown 总用 kill 是不安全的. 得益于 manners
- i18n 接口 i18n 接口非常轻量, 当 fmt.Sprintf 使就行
- 自定义信号 完全采用 os.Signal 接口, 安全关闭信号就是基于这个
- 延迟初始化 有些初始化工作需要在 main 执行时调用
- 子路由 按 http method 划分的子路由, 主路由只能由 main 函数调用
- 角色控制 字符串角色命名, 自动转化为 accessflags 支持的 interger
- 日志支持 引入 typepress/log, 支持 file 分割, email发送

- 数据库接口支持 引入typepress/db, 即便不需要也不必担心, 这是个轻量接口
- core 全局可访问的对象和 types 类型
- 基于 **Martini Injector** 的设计 这是最最重要的

这样列举起来, 貌似这个 Simple Server 貌似已经不轻量了. 不! 他依然是轻量的, 因为这些接口设计的很轻量, 当你不用他们的时候, 他们不会产生过多的消耗. 这些接口的代码都很短, 引入他们, 怎加不了多少代码空间. 应该可以看出仅仅是这些基础的功能已经形成了一个服务器框架.

这些都已经准备好了. 哦还有模板, 这个东西不打算默认引入, 各种口味难调.

模块化

这些很多都是独立的 package, 可以单独使用. 从 typepress org 可以看出, 模块以独立的 rep 出现. typepress 特别注意降低依赖, 写成独立 rep 是最基本的方法.

Go-Pages

熟悉 GitHub Pages 的读者, 看到 Go-Pages 已经想到 静态博客 这个词了. TypePress 从静态博客起步, 一点点迈进带数据库的博客系统. Github 的 Pages 功能已经提出了实用简洁的静态博客方案, jekyllrb 引擎为其提供强劲动力. Jekyll 给出了很好的文档规范, 可以直接借鉴其目录结构. Liquid 模板也有 Go 实现 Liquid Template Engine for Go. Go-Pages 尽可能兼容 Jekyll, 不能兼容的部分以后制作转换工具进行处理. 为此需要准备一些 package.

RootPath

rootpath 为多域名服务器绑定目录的 package. 效果上有点像 URLRewrite 的一个子集. 仅对 `http.Request.Host` 进行分析, 匹配成功设定相应的静态文件目录, 内容目录, 模板目录. 匹配失败拒绝访问或者不做任何操作. RootPath 让 Go-Pages 博客支持子域名(站群)或者 CNAME (绑定域名)支持.

static

static 在设定好的静态文件目录下, 响应 `URL.Path` 请求的静态文件, 尝试发送对应的 Gzip 预压缩文件 `pathto/URL.Path.gz` . 如果没有找到 static 不产生 404 , 它什么都不做. 不产生 404 有很多好处. 基于 Martini 的 Handler 一旦产生输出就会结束响应过程, 不产生 404 就可以继续进行处理, 比如自定义 404 页面, 比如进行动态 Gzip 压缩, 然后再交给 static 进行输出, 又或者那根本就不是个静态页面, 交给后续的 Handler 处理, 如果最终无法匹配, Martini 会执行 `http.NotFound` .

Liquid

Liquid 包提供了基本 Liquid 模板支持. Jekyll 对 liquid 其进行了一些扩展, 如果要完全兼容 Jekyll 是个庞大的工程. 但是, 有必要实现一些如 Global Variables 之类的. 用到的时候再分析.

特别的, Liquid 中的 `include` tag 需要使用者自己实现 `IncludeHandler` , 参见 `liquid.Configuration` 的接口.

MarkDown

轻量文本标记语言可以让书写者专注文章内容, 而不是为版式费神, 很适合书写博客. 有多种格式可选. Go-Pages 暂时支持最简单的 MarkDown 格式, 在前后端都要有所支持.

前端支持 MarkDown 的编辑器很多, markdown-editor 是比较简单的一个. blackfriday 是 Go 语言下的 MarkDown 解析器. 前端的博客文章编辑和提交这里不讨论了.

JingYes

前端 CSS 框架更是有太多选择, 当前比较受欢迎的当属 BootStrap 和 PureCSS. Go-Pages 使用 JingYes. 这里不再列举可能用到的其他前端库.

JingYes只支持现代的浏览器, 不过 html 源代码非常简洁, 可以很方便的改写成其它 CSS 框架.

TOML

配置文件采用 TOML 格式, 这里分析几个 `table` .

defalut

```
[defalut]
# 安全密匙, 请妥善保管, 切勿外泄.
# 受密匙的具体使用影响, 更改密匙可能会造成不可预计的破坏.
secret = ""
# 主站顶级域名
domain = ""
# 本地监听地址
laddr = ":80"
# 共享静态文件目录
static = "pages/defalut"
# 内容目录是独立的, 需要在 [[rootpath]] 中设置
content =""
# 共享模板文件目录
template = "pages/defalut/_layouts"
```

作为多域名博客系统有些 css,js,image 资源文件是可以共享的, static 目录起到这个作用. 但是, 可以预计, 很可能会把 MarkDown 文章源文件预渲染成 html 文件, 它也是静态文件, 他们所属的 base 目录是不同的. `static` package 不产生 404 的方式很好的解决了这个问题. Go-Pages 可以这样做(m 是 Martini 对象):

```
// 设定 defalut.static dir
m.Map(http.Dir(core.Conf["defalut.static"].String()))

m.Use(staticDefalutHandler) // defalut static 优先
m.Use(rootPathHandlerForDomain) // 这个一旦执行 root 就改变了
m.Use(staticHandlerForDomain) // 现在访问的静态文件就是站点的了
```

rootpath

```
[[rootpath]]
Flag    = 1 # 1 == FStatic, 每个域名都可以独立有静态文件
Root    = "pages/domain"
Pattern = "*"
Domain  = "localhost"
CategoryName = ["_site"] # Jekyll 的习惯用 _site 目录, 下述同理

[[rootpath]]
Flag    = 2 # 2 == FContent, 每个域名都有独立的 content
Root    = "pages/domain"
Pattern = "*"
Domain  = "localhost"
CategoryName = ["", "_posts"]

[[rootpath]]
Flag    = 4 # 4 == FTemplate, 尝试独立的 _layouts
Root    = "pages/domain"
Pattern = "?"
Domain  = "localhost"
CategoryName = ["", "", "_layouts"]
```

上述几个 package 给 Go-Pages 提供了最基础的动力. 流程也基本确定, coding...

解析器与舞台剧

好吧, 这是一个单章.

语法解析和编译原理是程序员的基础科目, 笔者却一直没有学好. 看着语法树跳来跳去的圈圈, 脑子里像有一群猴子在蹦跶. 一直想有机会补足这门功课, 为 TOML 写个解析器是个不错的选择. 因此 tom-toml 的解析是纯手工的. 作为一个新手笔者无法用正规准确的文字描述解析器的写法和原理, 因此本章用舞台剧来比喻解析器. 看官权当是看故事, 不必严格追究文法和用词.

本文指的是类 PEG 的方法, 这里有一篇翻译 解析表达文法. 简单的说 PEG 下的一切都是可确定的, 无二义性, 上下文无关, 无回溯的(线性时间). 这让我想到了舞台剧(事实是, 我先写完了 tom-toml 才发现用的是手工 PEG 的方法). 剧本是写好的, 场景, 台词, 演员, 结果都是固定的. 那么让我给你讲个大导演 TOM 导演一出舞台剧的故事.

汤姆的故事

吉它湖大剧院要办一场舞台剧, 这个任务当然是由大导演汤姆来做, 不然还有谁呢! 这天是周五下午三点, 汤姆接到了剧院下达的任务, 演一场舞台剧. 看完任务, 汤姆怨念骤起:

```
啥事儿都找我, 舞台剧反反复复都演了多少年了, 弄啥勒!
不中, 黑喽还得去斗地主咧, 快下班儿了, 时间紧任务急, 俩钟头弄完它.
杰森, 把咱勒临时演员都叫来, 有活儿了.
```

助理杰森抬起头

```
老板儿, 撒子事儿么? 你快说, 上次那个剧本的大括弧还没有写完呢.
```

汤姆

```
你胡扯啥, 不是叫你叫临时演员都来么!
```

杰森

```
没得问题! 100 分钟完成.
```

汤姆

```
拉到吧, 1 分钟, 只给你 1 分钟.
```

杰森扭头大喊

```
汤姆要发福利了....
```

殷特, 付乐得, 司琼在剧院门口已经蹲守三个月了, 不然还能去哪儿呢! 临时演员的职业操守就是等活儿和追讨工资, 蹲守是基本功. 杰森的声音让三人眼睛一亮, 下一秒就出现在汤姆面前, 此刻杰森的脖子还处于180度状态, 迷茫的望着门口.

```
我给恁仨说, 现在有个急活儿, 我直接念台词你们谁能演就言一声儿
```

汤姆掰了掰手指头

```
1
```

殷特和付乐得同时出声

```
我认识1, 我能演
```

司琼一脸漠然, 不懂啊. 汤姆傻眼了, 心说我正琢磨第一句说啥勒, 这俩伙当台词了, 算了, 我也懒的想了, 那就 1 吧, 这...

```
2
```

殷特和付乐得那个高兴啊

```
2 我也认识, 我能演
```

汤姆有点不高兴了, 这俩显摆啥咧, 认识个1,2就这德性

```
点
```

殷特茫然了, 付乐得心里高兴啊

```
点我也会, 我演吧
```

汤姆受不了了

```
12. 你都认识, 你学问不低啊, 不用说 12.3 你都认识了?
```

付乐得一拍胸脯

```
没问题, 12.3 我认识, 12.34567 我也能演
```

汤姆拿了张纸, 画个圈儿

```
中了, 这个角色给你了, 这是场次安排分剧本, 好好练练, 走吧
```

付乐得接分剧本扭头走了

```
咱接着走台词啊, 下个台词是...
"
```

司琼其实是他们三个学历最高的了, 研究生啊, 可惜书读的太多, 脑子就傻了, 只知道照本宣科, 从小就知道剧本台词都是以 " 开始的, 不然那就不是台词啊, 终于让琼斯等到了

```
汤姆导演, 这个是台词, 我会
```

汤姆一头黑线, 感情 12.3 就不是台词! 怒了

```
你啥意思, 我前面说了那么多, 那都不叫台词! 你@#%$#^%$&$%#@%$@#$
```

琼斯一声不吭, 支起耳朵, 紧锁眉头把汤姆的每一个字都记下

```
你以为你会了可多, 你不就认识个"
```

琼斯眉头一展

```
嗯, 汤姆导演, 你说的台词真好, 可标准了, 我都记下了, 保证演好
```

汤姆张大嘴巴足足90分钟才楞过神来

```
中, 就这吧, 给你拿好分剧本, 赶恁咧走吧
```

琼斯双手接过剧本, 鞠躬, 走人. 般特急了

```
导演给我也安排个活儿吧, 我都认识 0123456789 呢
```

汤姆随口说

```
那你就报开场倒计时吧
```

汤姆分了剧本, 忽略头才转回90度的杰森, 低头看表. 5点, 汤姆嘴角一扬走出了办公室.

PEG

PEG 的解析过程就像舞台剧, 固定的台词(待解析的文本), 固定的演员(token), 固定场景下有固定的演员和台词, 并且固定的转场. token 判定函数对得到的字符逐个判断, 例如当顺序流入

```
1234.567
```

直到字符 "4" 时, ItsInteger 和 ItsFloat 都认为可能认识这个 token, 都反馈 "可能是", SMaybe 状态. 出现了 ".", ItsInteger 返回"不认识" SNot 状态, ItsFloat 继续返回 SMaybe. 后续的字符都完毕 ItsFloat 都可识别, 都返回 SMaybe. 最后 ItsFloat 拿到 EOF 后确认认识, 返回 "确定是" SYes 状态. 识别 token 就是这么一个过程.

那么, 整个的解析流程就像舞台剧的场景, 每个场景是清楚会出现哪些 token 的. 以 TOML 语法为例, 开始场景命名为 stageEmpty, 可允许出现的 token 包括:

```
EOF             空文本也是允许的
Whitespace      白字符
NewLine         新行 LF,CR,LFCR,CRLF
Comment         # 注释
TableName       [tableName]
ArrayOfTables   [[arrayOfTableName]]
Key             键名
```

注: 上面的次序有效率问题, 甚至是必须的次序才能实现或简化代码. 周知开始场景和结束场景是相同的, EOF 出现在stageEmpty 中是理所当然的. 如果没有 token 被匹配, 那一定是语法错误. 如果匹配, 就进入下一个场景, 每个场景都有固定的 token 列表, 循环这个过程直到重回开始场景识别到 EOF. token 和场景变化可以这样描述

stageEmpty

```
 EOF             -> stageEnd
 Whitespace      -> stageEmpty
 NewLine         -> stageEmpty
 Comment         -> stageEmpty
 TableName       -> stageEmpty
 ArrayOfTables   -> stageEmpty
 Key             -> stageEqual
```

stageEqual

```
Whitespace      -> stageEqual
Equal           -> stageValue
```

stageValue

```
Whitespace      -> stageEqual
ArrayLeftBrack -> stageArray
String          -> stageEmpty
Boolean         -> stageEmpty
Integer         -> stageEmpty
Float           -> stageEmpty
Datetime        -> stageEmpty
```

stageArray

```
Whitespace      -> stageEqual
ArrayLeftBrack -> stageArrayWho
ArrayRightBrack-> stageArrayWho
String          -> stageStringArray
Boolean         -> stageBooleanArray
Integer         -> stageIntegerArray
Float           -> stageFloatArray
Datetime        -> stageDatetimeArray
```

stageStringArray

```
Whitespace      -> stageStringArrayComma
String          -> stageStringArray
ArrayRightBrack-> stageArrayPop
```

stageStringArrayComma

```
Whitespace      -> stageStringArrayComma
Comma           -> stageStringArray
ArrayRightBrack-> stageArrayPop
```

以此类推, 其中

```
为便于阅读, 上述定义省略部分新行和注释, 这不会影响理解.
Array 是可嵌套的, stageArrayWho 有多种实现方法, 需要专门的篇幅描述. 本文不讨论.
stageStringArray 也受嵌套影响, 肯定不能这么简单就得到 stageXxxxArray. 本文不讨论.
如果某个 token 在解析时做不到验证完整性, 可以放到生成 Toml 时再检查.
```

注**:** 在本新手眼里 **Array** 的嵌套被当作左递归的一种**,** 理论上 **PEG** 要求消除左递归文法**,** 先手工硬编码解决这问题吧**.**

完全手工构造场景变化表是比较痛苦的, 可以把 token 匹配和文法合法性检查分开, 减省 stage 的数量. 比如 stageStringArrayComma 就可以减省, 留给其他代码处理.

你会发现不同语言实现的 PEG, 在表达式文法和用词上甚至不一致. PEG 确实没有规定确切文法用词, PEG 关注的是解析中的逻辑关系.

ABNF

BNF 是巴科斯范式, 英语：Backus Normal Form 的缩写, 也被称作 巴科斯-诺尔范式, 英语: Backus–Naur Form. Backus 和 Naur 是两位作者的名字. 必须承认这是一项伟大的发明, BNF 开创了描述计算机语言语法的符号集形式. 如果您还不了解 BNF, 需要先 Google 一下. 随时间推移逐渐衍生出一些扩展版本, 这里直接列举几条 ABNF RFC5234 定义的文法规则.

PS: 后续代码实现部分, 早期的代码思路不够精简, 在现实中很难应用. 后期又做了纯规则的实现, 请读者选择性阅读, 以免浪费您宝贵的时间

```
; 注释以分号开始
name      = elements          ; name 规则名, elements 是一个或多个规则名, 本例只是示意
command   = "command string" ; 字符串在一对双引号中, 大小写不敏感
rulename = %d97 %d98 %d99    ; 值表示 "abc", 大小写敏感, 中间的空格是 ABNF 规则定界符
foo       = %x61              ; a, 上面的 "%d" 开头表示用十进制, "%x" 表示用十六进制
bar       = %x62              ; b, "%x62" 等同 "%d98"
CRLF      = %d13.10           ; 值点连接, 等同 "%d13 %d10" 或者 "%x0D %x0A"

mumble    = foo bar foo       ; Concatenation 级联, 上面定义了 foo, bar, 等同区分大小写的 "aba"

ruleset   = alt1 / alt2       ; Alternative 替代, 匹配一个就行, 以 "/" 分界
ruleset   =/ alt3             ; 增量替代以 "=/" 指示
ruleset   = alt1 / alt2 / alt3 ; 等同于上面两行
ruleset   =  alt1 /           ; 你也可以分成多行写
             alt2 /
             alt3

DIGIT     = %x30-39           ; Value range 值范围, 用 "-" 连接, 等同下面
DIGIT     = "0" / "1" / "2" / "3" / "4" / "5" / "6" / "7" / "8" / "9"

charline = %x0D.0A %x20-7E %x0D.0A ; 值点连接和值范围组合的例子

seqGroup = elem (foo / bar) blat ; Grouping 分组, 一对圆括号包裹, 与下面的含义完全不同
seqGroup = elem foo / bar blat   ; 这是两个替代, 上面是三个级联

integer   = 1*DIGIT             ; Repetition 重复, 1 至 无穷 个 DIGIT
some      = *1DIGIT             ; 0 至 1 个 DIGIT
someAs    = 0*1DIGIT            ; 等同上一行
year      = 1*4DIGIT            ; 1 至 4 个 DIGIT
foo       = *bar                ; 0 至 无穷 个 bar
baz       = 3foo                ; 3 次重复 foo, 等同 3*3foo

number    = 1*DIGIT ["." 1*DIGIT]    ; Optional 可选规则, 用中括号包裹, 等同下面两种写法
number    = 1*DIGIT  *1("." 1*DIGIT)
number    = 1*DIGIT 0*1("." 1*DIGIT)
foobar    =  baz ; prose-val 用尖括号括起来, 值就可以包含空格和VCHAR
                                    ; 范围是 *(%x20-3D / %x3F-7E)
```

上面的描述用的也是 ABNF, 事实上这些文字就源自 RFC5234 规范. 级联规则就是一个顺序匹配的序列, 好比 Seq 顺序规则或者叫 And 规则. 替代好比 Or 规则或者叫 Any 规则.

四则运算表达式

现在我们尝试一下四则运算表达式的 ABNF 写法. 我们从人脑运算方式逐步推演出正确的写法. 周知四则运算会包含数字和运算符还有括号.

```
; 错误写法一
; Expr 表示要解决的问题, 四则运算规则
Expr = Num /                    ; Num 表示数字, 仅仅一个数字也可以构成Expr
       Num   Op   Expr /        ; Op  运算符
       "("   Expr  ")" /        ; 括号会改变 Expr 运算优先级
       Expr  Op   Expr          ; 最复杂的情况

Op   = "+" / "-" / "*" / "/"  ; 运算符的定义
Num  = 1*(0-9)                  ; 最简单的正整数定义
```

上面的写法模拟人脑做四则运算的习惯, 很明显绝大多数解析器都无法使用这个规则. 因为出现了左递归. "最复杂的情况" 这一行中 Expr 出现在规则的最左边, 这将导致解析器递归, 造成死循环. 虽然可以把解析器设计的复杂一些, 解决左递归, 但这会使解析器很复杂, 并造成效率低下, 时间复杂度陡增, 所以通常要求写规则时就消除左递归.

继续分析推演. 消除左递归一般通过因式分解(factor)或者引入新的规则条款(terms)解决. 通过 factor 或者 term 解除左递归发生的可能性, 好比多绕几个圈子, 多给解析器几条路, 让解析器绕过死循环的路径. 下面加上了 Repetition 重复规则. 我们先按照人脑思维, 乘法除法优先的顺序来写.

```
; 错误写法二
Expr   = Term   *Mul / ; Mul 是乘法, *Mul 表示可能有或没有, Term 就是要绕的圈子了.
         Term   *Quo   ; 除法和乘法一样, Term 这个圈子其实表示的还是 Expr.
Term   = Factor *Add / ; 一个圈子明显不行, 再绕个圈子 Factor,
         Factor *Sub   ; 这两行描述加减法, 逻辑都没错吧, 都是可能有, 也可能没有

Factor = Num /         ; 绕再多圈子总是要回来的, 数字总要有吧
         "(" Expr ")"  ; 括号的运算总要有吧

Add    = "+" Term      ; 一旦出现运算符, 后面一定会有后续的表达式吧
Sub    = "-" Term
Mul    = "*" Factor
Quo    = "/" Factor
Num     = 1*(0-9)
```

看上去会发生左递归么? 不会, 怎么绕你都不会死循环, 因为 Factor 的第一条规则 Num , 给绕圈圈一个结束的机会. 这个叫终结符. 但是这个写法是错误的. 你可以在脑子里模拟下 `1+2-3`, 到 `-` 号的时候就解析不下去了. `1+2` 被

```
Term   = Factor *Add
```

匹配了, 但是后面还有 `-` 号, 被匹配的是加法规则

```
Add    = "+" Term  ; 最后一个又回到 Term
```

但是 Term 无法匹配减号, Term 推演规则中没有以减号开头的. 你说重头来不就行了? 不行, 解析器执行的规则是找到一条路可以一直走下去, 如果走不动了, 就表示这条规则匹配完成了, 或者失败了. 减号来的时候, 如果假设解析器认为 `1+2` 已经走完, 减号来的时候还是要从 Expr 开始, 不能直接从 Sub 开始, 开始只能有一个, 从 Expr 开始推导不出首次就匹配 "-" 号的. 所以 `1+2-3` 没有走完, 解析进行不下去了.

那上面的问题出在哪里呢? 问题在:

终结符在推导循环中不能首次匹配

问题的逻辑是:

可以穷举开始和结尾, 不能穷举中间过程.

解决方法是循环或者递归:

在循环和递归中已经没有明确的开始,头尾相接就没有头尾了,没有头尾也意味能一直绕下去

综合这三句话, 我们解决问题的方法也就出来了:

```
1. 引入Term, Factor 消除左递归
2. 要给终结符在循环中首次匹配的机会或者说不阻断循环的进行
```

终结符就是推导循环到了最后, 不包含推导循环中的其他规则名. 再来符号就是新的, 要重头开始. 有终结但无法继续重头开始, 圈子绕不下去了.

继续推演, 我们先确定终结符. 我们用个小技巧, 按优先级合并运算符.

```
; 正确写法
Expr   = Term   *Sum   ; 继续绕圈子, *Sum 有或者没有, 先写求和是有原因的
Term   = Factor *Mul   ; 再写乘积, *Sum 不匹配, 就尝试乘积
Sum    = SumOp  Term   ; 求和的运算, 有运算符必定要有后续表达式
Mul    = MulOp  Factor ; 乘积的运算,
Factor = Num /         ; 引向终结
         "(" Expr ")"  ; 括号永远都在

Num    = 1*(0-9)      ; 数字, 这可以是独立的终结符
SumOp  = "+" / "-"    ; 加或者减, 可以叫做求和, 小技巧
MulOp  = "*" / "/"    ; 乘或者除, 可以叫做乘积
```

把这两种写法左右排列, 看的更清楚

```
; 错误写法                   ; 正确写法
Expr   = Term   *Mul /     ; Expr   = Term   *Sum   ; 蛇头
         Term   *Quo       ; Term   = Factor *Mul   ; 蛇头
Term   = Factor *Add /     ; Sum    = SumOp  Term   ; 咬蛇尾
         Factor *Sub       ; Mul    = MulOp  Factor ; 咬蛇尾

Factor = Num /             ; Factor = Num /
         "(" Expr ")"      ;          "(" Expr ")"

Add    = "+" Term          ; SumOp   = "+" /
Sub    = "-" Term          ;           "-"
Mul    = "*" Factor        ; MulOp   = "*" /
Quo    = "/" Factor        ;           "/"
Num     = 1*(0-9)          ; Num     = 1*(0-9)
```

你应该发现了, 主要区别是: 运算符和后续的 Expr 的结合处理方式不同. 左侧的规则是: (数字,运算符,数字) 然后还想找 (数字,运算符,数字). 右侧的规则是: (数字,运算符) 然后继续 (数字,运算符), 最后找到终结.

左侧规划了一条既定的有终结路线, 走不了几步就终结了. 蛇头没咬到蛇尾, 咬到七寸了. 右侧规划了一条蛇头咬蛇尾的循环路线, 循环中所有的规则名都有机会匹配.

解析器

这是早期思路所写的代码, 事实上我自己用起来也很不舒服, 也一直没有放出来, 就当做失败的例子吧

ABNF 具有很强的表达能力, 这里以 ABNF 为基础分析要分离出的规则元素. 终结符和非终结符, 可以这样描述

1. atom 终结符, 就是个对输入字符进行判断的函数
2. term 非终结符, 可能有多个或多层 term/atom 组合, 也可用 group 这个词
3. factor 抽象接口, term 和 atom 的共性接口, 代码实现需要抽象接口

用 group 替换 term 在语义上也是成立的, 一个独立的 term 也可以看作只有一项规则元素的 group.

元素关系可以分

1. Concatenation 级联匹配
2. Alternation 替代匹配

atom 可以看作只有一项的级联, group 需要选择两者之一. 那么 group 需要可以增加规则元素的接口

1. Add(factor)

在做解析的时候常采用循环的方法, 某个循环结束后会产生两个状态

1. ok 解析是否成功

2. end 解析是否结束

任何时候遇到 end, 无论处于那一级循环中, 都要终止解析. 如果不是 end, 那么依据元素关系和解析是否成功进行判断, 决定尝试匹配下一个规则元素或者返回 !ok, !end.

其他

1. 给 term/group, atom/factor 命名或设定 id
2. 设置 term/group, atom/factor 的重复属性 repeat.

综合上述分析大致的接口设计如下

```
type Factor interface {
    /**
    每个 Factor 都有一个唯一规则 ID
    Grammar 的 id 固定为 0.
    Atom, Group 自动生成或者设定. 自动生成的 ID 为负数.
    */
    Id() int

    /**
    返回 Factor 的风格. 常量 KGrammar / KGroup / KFactor.
    这里简单用 int 类型区分
    */
    Kind() int

    /**
    Mode 返回 Factor 所使用的匹配模式
    Atom 总是返回常量 MConcatenation.
    */
    Mode() int

    /**
    匹配脚本
    参数: Scanner 是个 rune 扫描器, Record 用于记录解析过程.
    返回值:
        ok      是否匹配成
        end     是否终止匹配,  事实上由 Record 决定是否终止匹配
    */
    Process(script Scanner, rec Record) (ok, end bool)
}

// 语法接口也基于 Factor
type Grammar interface {
    Factor
    /**
    生成一个 Term 过渡对象.
    初始重复 1 次, 1*1.
    初始匹配模式为 MConcatenation.
    参数 id 如果 <= 0 或者发生重复, 那么自动生成负数 id.
    */
    Term(id int) Term

    // 为 Grammar.Process 设置最初规则.
    Start(rule ...Factor) Grammar

    // 设置为 Concatenation 匹配模式
    Concatenation() Grammar

    // 设置为 Alternation 匹配模式
    Alternation() Grammar
}

/**
term 是个中间件, 最终要转化为 Group/Factor
*/
```

```
type Term interface {
    Factor
    // 为 Term 命名. 不检查名称唯一性.
    Named(string) Term

    /**
    设置 Repeat, 参数 a, b 对应 ABNF 的 repeat 定义 a*b.
    如果 b < a, 把 b 作为 0 处理.
    */
    Repeat(a, b uint) Term

    // 转为 Group
    Group() Group

    // 由 Atom 转为 Factor
    Atom(atom Atom) Factor
}

/**
Group 具有 Add 方法
*/
type Group interface {
    Factor
    // 设置为 Concatenation 匹配模式
    Concatenation() Group
    // 设置为 Alternation 匹配模式
    Alternation() Group
    /**
    添加一组规则.
    如果没有通过检查返回 nil
    */
    Add(rule ...Factor) Group
}
```

从中可以看出, Term 是个过渡接口, 设计这个的原因是:

ABNF 文法中的规则定义和程序中的类型定相似, 次序无所谓, 只要有定义. 很明显实现解析器的时候, 这些元素是以变量的形式存在的, 我们需要先生成变量, 然后在进行关系组合. 笔者实现了一个

```
// 丑陋的 Term 表现四则运算
// g 是个 Grammar 对象, 起个名字叫 Arithmetic
g := New("Arithmetic")

// 先生成规则元素, Atom 的参数 id 是预定义的常量
expr := g.Term().Named("Expr").Group()
end := g.Term().Named("EOF").Atom(IdEof, ItsEOF)
num := g.Term().Named("Num").Atom(NUM, ItsNum)

// GenLiteral 是个辅助函数函数, 生成字符串匹配 Atom
C := g.Term().Named("(").Atom(LPAREN, GenLiteral(`(`, nil))
D := g.Term().Named(")").Atom(RPAREN, GenLiteral(`)`, nil))

term := g.Term().Named("Term").Group()
sum := g.Term().Zero().Named("Sum").Group()
factor := g.Term().Named("Factor").Group().Alternation()
mul := g.Term().Zero().Named("Mul").Group()

sumOp := g.Term().Named("SumOp").Group().Add(
    g.Term().Named("+").Atom(ADD, GenLiteral(`+`, nil)),
    g.Term().Named("-").Atom(SUB, GenLiteral(`-`, nil)),
).Alternation()

mulOp := g.Term().Named("MulOp").Group().Add(
    g.Term().Named("*").Atom(MUL, GenLiteral(`*`, nil)),
    g.Term().Named("/").Atom(QUO, GenLiteral(`/`, nil)),
).Alternation()

nested := g.Term().Named("Nested").Group().Add(C, expr, D)

// 组合规则元素
g.Start(end, expr)

expr.Add(term, sum)
term.Add(factor, mul)
sum.Add(sumOp, term)
mul.Add(mulOp, factor)

factor.Add(num, nested)

// 这里省略了 script 和 rec 的生成
g.Process(script, rec)
```

Term 的出现, 虽然逻辑上完整了, 代码写出来看上去很丑陋. 看来只有通过辅助函数简化代码了.

手工至上

连续两章学习解析器, 事实上笔者自己尝试实现了一个基于 ABNF 的解析器雏形. 然而由于采用了递归的匹配方式, 最终的测试结果让人非常沮丧, 和 go 官方提供的 json 包对比解析速度慢几十倍. go 官方的 json 包是纯手工代码实现的. 采用大家常常听到的先词法分析后语法分析的方法, 事实摆在眼前, 这种手工的方法真的是最快的. 同样的方法我们在 go 的标准库中可以找到多处, 各种教课书中讲的解析相关知识根本就没用上, 效率有目共睹. 直接看相关源代码就能了解细节.

手工代码构造的先词法分析后语法分析的解析器是最快的

再接再厉

Zxx 是很偶然的一次 Q 群聊天玩笑的产物, 到目前为止这依然是个设想(玩笑). 好在新的 zxx abnf 产生了. 这次尝试了另外的思路, 到目前为止感觉还不错.

如前文所述, 可以从 ABNF 规范中抽离出独立的匹配规则, 下文用 R 代表一个规则.

1. Zero 规则对应 R* 匹配零次或多次
2. Option 规则对应 R{0,1} 匹配零次或一次
3. More 规则对应 R{1,} 匹配一次或多次
4. Any 规则对应多个规则匹配任意一个
5. Seq 规则对应多个规则被顺序匹配
6. Term 所有规则是有 Term 组成的.

这些只是基本的匹配规则逻辑. 毫无疑问, 文法解析是从一个字节一个字节进行的, 前文的实现也是这么思考的. 现在换个角度考虑问题:

```
字符串也好, 叫做 Token 也罢, 真正要做的是判断一个 Token 是否满某个条件.
```

我们知道解析的最小单位是 Token, 我们个 Token 加一个成员方法

```
// Has 返回 token 等于 tok 或者包括 tok
func (token Token) Has(tok Token) bool

// 这里截取部分 Token 定义
const (
	EOF Token = iota
	Type // 分类标记
	// 预定义类型
	BYTE
	STRING
	UINT
	INT
)
```

那么 Type.Has(INT) 的值就为 true.

即 Token 的 Has 方法为前述的 ABNF 规则提供了最底层的判断. Token 的类型不再重要, Has 保证了一切. 现实中可从扫描器得到 Token. 而 zxx abnf 只关心相关的规则定义.

列举下相关定义, 有些注释省略, 完整代码在 zxx abnf:

```
// Flag 表示规则匹配 Token 后的状态
type Flag int

const (
	Matched  Flag = 1 << iota // 匹配成功, 消耗掉一个 Token
	Standing                  // 规则成立, 可以继续匹配 Token
	Finished                  // 规则完整, 不再接受匹配 Token

	// 下列标记由算法维护, Match 返回值不应该包含这些标记.
	Handing // 正在进行处理中(Match), 可用来检查发生递归
```

```
    Cloning // 正在进行克隆
    Custom  // 通用标记位, 包括更高的位都由 Match 自定义用途,
)

type Rule interface {
    // Match 返回匹配 tok 的状态标记. 实现必须遵守以下约定:
    //
    // 返回值为下列之一:
    //
    //    0
    //     Matched
    //     Matched|Standing
    //     Matched|Finished
    //     Finished
    //
    // 自动重置:
    //
    // 规则状态为 0, Finished, Matched|Finished 时自动重置, 可接受新的匹配.
    //
    // EOF 重置: 当最终状态为
    //
    //    Matched 最终状态是不确定完整匹配.
    //    Matched|Standing 最终状态是完整匹配.
    //
    // 时使用 Match(EOF) 重置规则并返回 Finished, 其它状态不应该使用 EOF 来重置.
    //
    // 末尾完整测试:
    //
    // 类似 Seq(Term(XX),Option(YY),Option(ZZ)) 规则, 单个 XX 也是合法的,
    // 但是由于 Option 的原因, 匹配单个 XX 的状态为 Matched,
    // 因此再匹配一个不可能出现的 Token, 可以测试规则是否完整.
    //
    Match(tok Token) Flag

    // Bind 应只在递归嵌套规则变量时使用, 先声明规则变量后绑定规则.
    // 其他情况都不应该使用 Bind.
    Bind(Rule)

    // Clone 返回克隆规则, 这是深度克隆, 但不含递归.
    // 递归规则在 Match 中通过判断 Handing 标记及时建立的.
    Clone() Rule

    // IsOption 返回该规则是否为可选规则.
    // 事实上除了 Option 是明确的可选规则外, 其它组合可能产生事实上的可选规则.
    IsOption() bool
}

// Term 用来包装 Token

// Term 产生任一 Token 匹配规则. Match 方法返回值为:
//
//    0
//    Matched | Finished
//    Finished     当 EOF 重置或者 tok == nil
//
func Term(tok ...Token) Rule
func Option(rule Rule) Rule
func Once(rule Rule) Rule

// More 产生重复匹配规则.
// rule 必须初次匹配成功, 然后当 rule 匹配结果为 0, Finished 时尝试 sep 匹配,
// 如果 sep 匹配成功则继续匹配 rule.
//
func More(rule, sep Rule) Rule

// Any 产生任一匹配规则.
// 不要用 Any(rule, Term()) 替代 Option, 那会让 IsOption() 不可靠.
func Any(rule ...Rule) Rule

// Seq 产生顺序匹配规则
func Seq(rule ...Rule) Rule
```

你可能注意到其中没有 Zero 规则, 因为不需要它, Flag 的 Finished 隐含的兼容了 Zero 规则. 只有 Flag(0) 才表示失败, Finished 虽然表示 Token 未被匹配, 但是规则是成立的, Token 由后续规则继续匹配好了.

先写这么多**,** 就 **5** 个规则**,** 写多了反而添乱

Router

WEB 开发离不开 Router. 通常 Router 负责对 HTTP Request URL 进行分析, 匹配到对应的处理对象. URL 可以分为三部分 Host, Path, QueryParams. 在官方包提供的 http.Request 对象中有对应的字段. 以前作者没有关注路由具体实现, 只是拿来用. 很偶然发现一个路由评测项目 go-http-routing-benchmark. 尽管路由的开销很低, 仍旧很惊异不同路由有这么大差异, 那自己也实现一个吧.

Rivet

于是 Rivet 诞生了. Rivet 学习了 httprouter 的方法, 用前缀树(Trie)管理路由节点, 这是提高匹配速度的关键. 另外作者发现事实上:

- 带参数的 URL.Path 很普遍, 字符串参数可能被转换类型.
- 路由处在处理请求的前端, 这期间应用应该有机会拒绝请求.
- Host 路由应当被支持.
- Martini 的注入方式确实方便.
- 路由应该可被独立使用, 而不是和框架强耦合

Rivet 满足了这些需求, 而且性能非常可观.

Module

既然 Rivet 采用了注入方式, 那应该可以开发只使用 Go 自带 pkg 与框架无关的独立模块, 以便应用选取不同的框架. 当然事实上选用支持注入的框架是最方便的.

mod 就是这样的尝试. 目前, 作者也不知道能有多少模块可以采用这种开发方式, 我会继续尝试.